Reducing Component Contract Overhead by Offloading Enforcement

[Extended Abstract]

Kostadin Damevski
Virginia State University
1 Hayden Dr.
Petersburg, Virginia, 23806, USA
kdamevski@vsu.edu

Hui Chen
Virginia State University
1 Hayden Dr.
Petersburg, Virginia, 23806, USA
hchen@vsu.edu

Tamara L. Dahlgren
Lawrence Livermore National Laboratory
Livermore, CA, 94550, USA
dahlgren@llnl.gov

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Programming by contract

General Terms

Design, Reliability

Keywords

Runtime Checking, Contract Offloading

1. INTRODUCTION

In recent years, component technology has been a successful methodology for large-scale commercial software development. It is also becoming increasingly popular for large-scale scientific computing in helping to tame the software complexity required in coupling multiple disciplines, multiple scales, and/or multiple physical phenomena. The Common Component Architecture (CCA) [1] is a component model that was designed to fit the needs of the scientific computing community by imposing low overhead and supporting parallel components. CCA has already been used in several scientific domains, creating components for large simulations involving accelerator design, climate modeling, combustion, and accidental fires and explosions [6].

Executable interface contracts present a way to better specify interfaces through which components interact, and to dynamically verify whether this specification is adhered to by an executing application. In this way, composability of applications and the quality of software components is improved. Contracts uncover both errors which would have resulted in the application being killed by the OS and errors where the application produces incorrect results, which are more difficult to trace. Although they present a performance overhead, in order to be most effective contracts need to be executed during all uses (and through a variety of parameters) of a software component, including in production scenarios. Contracts are used in many computational domains ranging from embedded systems to high-performance computing. High-performance computing applications are particularly performance overhead averse, forcing contracts for scientific components to suffer from reduced use due to the performance cost. Contracts have been developed for the CCA component software paradigm which is used in scientific computing [3], but their user isn't prevalent in this community.

We posit that contracts can be just as effective in determining the existence and source of application errors if they are enforced at a different time from the execution of the application. To realize this, we propose a system to reduce the overhead of contract enforcement in CCA component applications by offloading it to a separate space and time. Instead of taking up precious application resources, contracts are offloaded to and enforced by a specialized component that performs this task when or where computational resources are available with minimal interference to the application itself. By offloading contracts in this manner we introduce a separation between the time when specific contracts are encountered by an application, when they are checked, and when the application is informed of possible contract violations. This introduces a few new scenarios in the appearance of contract violations to the executing application. In addition, due to certain optimizations in our design, a possibility (though relatively small) is introduced that a contract may not be checked altogether because of the inability of the contract enforcement to keep up with the executing application. However, since the missed contract is randomly chosen, based on system load and CPU scheduling, there is a good chance that it will be checked in subsequent executions of the same application. In this paper, we argue that our approach to enforcing executable contracts is a feasible way to apply this software engineering technique to overhead-averse scientific applications.

We organize the discussion of our offline contract enforcement system as follows. Section 2 contains background and discussion of the problem. In Section 3 we describe and analyse our design, while in Section 4 we discuss the related work. Finally, we finish with conclusions and future work of the project in Section 5.

CBHPC'09, Nov. 15-16, Portland, OR, USA

2. BACKGROUND

A CCA component consists of one or more ports, and a port is a group of method-call based interfaces. A CCA port is represented by an interface, which is specified through the Scientific Interface Definition Language (SIDL). A SIDL specification is compiled into glue code that is later compiled into an executable together with the user-provided implementation code. The prevalent way of compiling SIDL is by using the Babel compiler [4]. Babel has the capability of compiling SIDL to bindings for several popular programming languages (C++, Java, Python, Fortran77 and Fortran90), which allows creating applications by combining components written in any of these languages. Babel has a large and growing user community and has become a cornerstone technology of the CCA component model.

Executable interface contracts in Babel consist of method preconditions and postconditions, assumed to hold true before and after the execution of the actual method respectively. A failed contract indicates a runtime unrecoverable error in the application. Some contracts are simple, such as checking a pointer for null, and enforceable in constant time. Others may involve checking whether an entire array of values is within a certain tolerance of zero, or performing a more complex computation, and can be linear time or slower. Babel offers an expressive contracts grammar that includes support for logical operations (e.g. *and*,*or*,*if and only if* and others), function calls, as well as a number of predefined array functions (e.g. *sum*, *min*, *max* and others).

A user-specified limit in the contract enforcement overhead, where contracts encountered after the limit is reached are ignored, is one of the approaches taken in order to make contracts more attractive to the high-performance computing domain [2]. This approach presents several policies (adaptive, random, periodic etc.), some of which enforce a part of the available contracts up to a given overhead limit (e.g. 20% of application runtime).

In order to be able to express and enforce contracts in a complete manner, and be able to enforce them in deployment scenarios, one has to contend with a possibly large overhead. In this work, we attempt an approach that would greatly reduce the overhead to the application by offloading complex and time consuming contracts.

3. DESIGN

Our system offloads the enforcement of contracts to a separate contract enforcer component, adding very little overhead on the running application. The contract enforcer component contains a queue of contracts that need to be executed, together with a low priority level thread that processes the contracts. This thread executes when the processor is free [1] and checks as many contracts as it can within its timeslice. When a contract violation is detected, the contract enforcer component informs the application who, in turn, stops execution and informs the user at the next opportune moment - usually, at the beginning or end of a method. This happens regardless of whether the component that incurred the violation is still active in the application. In order to provide a bound the memory taken up by the contracts queue, we use a circular queue where the most recent contracts may overwrite the ones which have been stored the longest. We accept the fact that some contracts may be left unchecked, because of the inability of the thread to keep pace when the system is busy. An overview of this design is also shown in Figure 1. When an application exists successfully (without a noticeable exception), we offer the user the possibility of checking its remaining contracts in the queue. In order to differentiate between queued contracts that belong to separate applications, we use a two-level circular queue where the 1st-level circular queue holds the heads to the 2nd-level circular application queues.

A side-effect of offloading contracts is that the reporting of contract violations to the application may be delayed. One can differentiate between three types of delayed contract violations based on their arrival time: 1) the application is still executing, 2) the application has finished executing, or 3) the application encountered an error. If the application is still executing when the contract violation is reported to it, it will report this occurrence by (usually) throwing an exception. The extra processing performed by the application in the period of time until it receives the contract violation should be discarded. On the other hand, if the application has finished executing and there is a contract violation to report, we attempt to deliver this to the user using some mechanism, such as through a log or user interface message. The third case, when a new error was encountered while the offloaded contracts are being checked, is due either to the same application inconsistency as the contract violation itself or due to a separate reason altogether. In either case, we take a similar approach as before and find a way to report the contract violation to the user.

As mentioned earlier, we use a circular buffer in order to limit the memory consumption of the contract enforcer component. Depending on the size of this buffer, the amount of contracts offloaded, and the speed of enforcement, it is possible that newer contracts overwrite unchecked contracts in the circular buffer resulting in missed contracts. We accept this loss as part of our decisions to offload contracts and limit their memory storage while minimizing interference with the running application. As long as missed contracts are infrequent, it is unlikely they will impact the system significantly since contract violations usually occur relatively infrequently in well tested software.

3.1 Feasibility Analysis

A component application can be divided into a series of method calls $m_x, x \in 1...n$, each of them belonging to an individual component. Each method can have a number of contract clauses enforced at its entry and its exit, $pre_{x,1}...pre_{x,m}$ and $post_{x,1}...post_{x,k}$, each of them internally consisting of one or more assertions. The runtime of the overall application is the sum of all its methods and contracts[2],

$$\sum_{i=0}^{n}\sum_{j=0}^{m}\sum_{k=0}^{l} m_i + pre_{i,j} + post_{i,k}$$

Each contract clause performs computation on some data $ccomp_d$ in order to determine whether the condition it imposes is met. When offloading the contract enforcement, the application no longer suffers the cost of contract enforcement while it incurs a new type of a performance overhead, which is the cost of performing an in memory copy of the

[1] In a single core system it would be when the application is waiting on I/O, while if more processing units are available this thread may execute continuously.

[2] For simplicity, in this cost equation we are neglecting to account for the driver method (or main()).

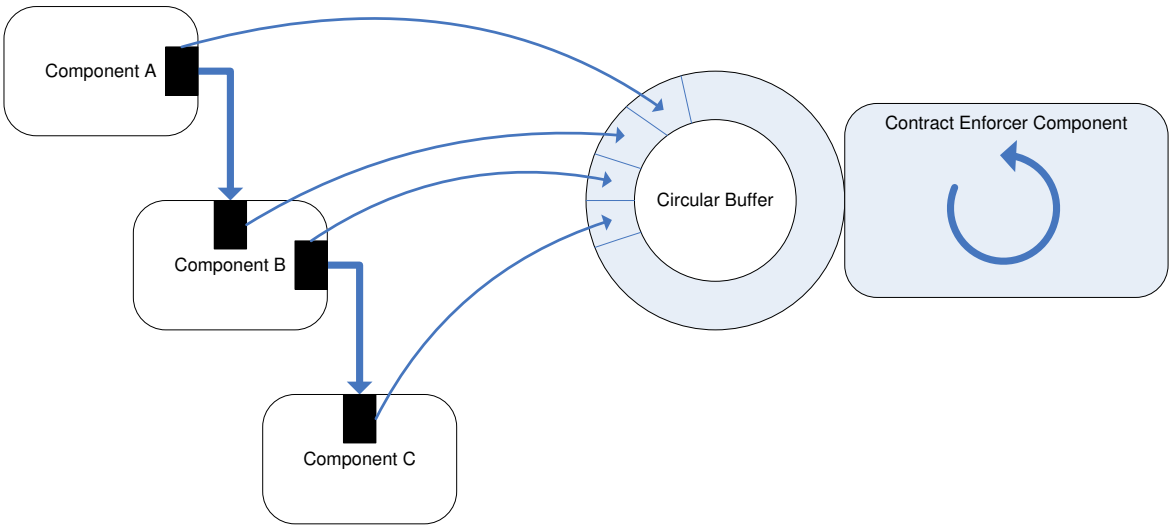

Figure 1: Diagram of the system to offload contract enforcement to a separate contract enforcer component.

data needed to enforce the contracts. A copy is necessary in this case, as the application may modify the data that should be checked by a contract clause. We represent the time it takes to copy the contract clause as $copy_d$. While this operation's complexity is linear ($O(n)$), it is relatively inexpensive in practice for data that is able to fit into memory. In order to have a performance gain, the cost of copying contract clauses needs to be less than the cost of enforcing them (i.e. $copy_d < ccomp_d$). It naturally follows that larger the computational complexity computation of a contract clause, the bigger the performance benefit of offloading it. For clauses that do very little computing (e.g. constant time complexity) offloading may even be more expensive than checking the contract and would adversely impact performance. Therefore, we limit the offloading to contract clauses which are linear time or more in cost.

Babel's contract system tags each contract in terms of its computational complexity. Using Babel, we execute and benchmark the following method that calculates vector dot-product and includes contracts:

```
double vDot(in array<double> u, in array<double> v,
            in double tol)
  throws
    sidl.PreViolation, sidl.PostViolation;
  require
    not_null_u: u != null;
    not_null_v: v != null;
    same_size:  size(u) == size(v);
    non_negative_tolerance: tol >= 0.0;
  ensure
    no_side_effects: is pure;
    same_uv: nearEqual(u, v, tol) implies
               (result >= 0.0);
    zero_case: (irange(u, 0.0, tol) and
                  irange(v, 0.0, tol))
                    implies
                      nearEqual(result, 0.0, tol);
```

The *vDot* method has a number of preconditions, listed under the *requires* SIDL keyword, and a number of post-conditions, under the *ensures* keyword. The postconditions contain a number of linear time contracts, which are defined using Babel array built-in functions. The *nearEqual* function determines whether the elements of two arrays are within a given tolerance of each other, while the *irange* function determines whether all elements of an array are within a given range. A visualization of the contract overhead in executing this method compared to offloading it is shown in Figure 2. To offload the contract, the data needs to be copied into the circular buffer, which still presents an overhead. This benchmark was executed on a single machine with an 2 GHz Intel Core 2 Duo processor and 1GB of RAM, with vectors containing 5000 elements, and averaged from 100 executions. We observe that there is a performance improvement in offloading the contracts that would grow further with larger data sizes and frequent invocations of this method in an application. We also observe that the cost of copying the data to offload it is significant and only worthwhile for complex contracts that include more that one linear computation.

4. RELATED WORK

We are unaware of any previous work discussing offloading contracts outside of the application's context, as described in this paper. The efforts of Dahlgren to establish the contracts facility for the CCA component model [3], and, subsequently, to provide an adaptive enforcement strategy that limits the cost of contracts [2] forms the foundation for the work in this paper. Several other approaches have been taken in reducing the overhead of contracts.

Liblit et al. [5] propose a system to gather and analyse program invariants (assertions) across a large group of distributed program executions; an example application given is that of Microsoft Office, where a large user community exists. In order to impose only negligible overhead to individual users, a clever scheme is designed that distributes assertion checks across many executions in a way that only few assertions are checked per run, while providing overall good coverage of each assertion.

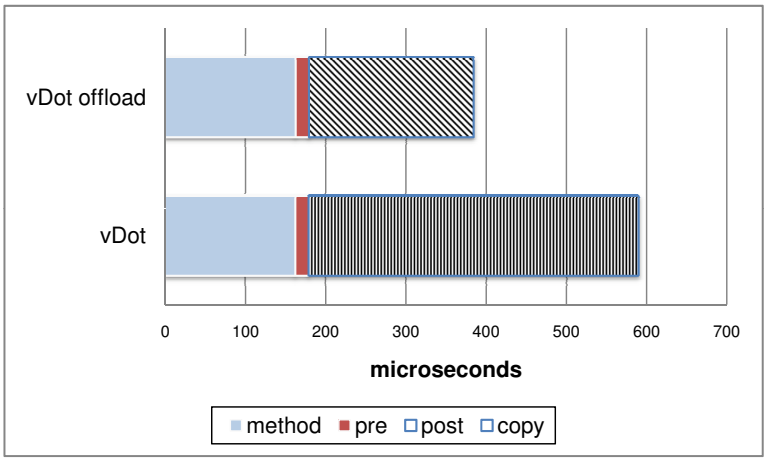

Figure 2: Comparison of executing the *vDot* method with and without contract offloading.

An approach in software engineering of embedded systems performs post processing of application's log data in order to detect errors [7]. By integrating two tools: the Avrora sensor network simulator, and the MaC (Monitoring and Checking) program verification tool the authors offer an approach to improve the quality of software for wireless sensor networks. The Avrora simulator, a well established tool in this field, is used to collect runtime data, which is analysed by MaC to determine whether the application's guarantees are met.

5. CONCLUSIONS AND FUTURE WORK

In this paper we discussed our design of a system to offload executable interface contracts in order to reduce their overhead and increase their acceptance in the high performance computing community. We argue that such a system would be beneficial in that it improves application performance while still enforcing the majority of contracts. The new sets of behaviors introduced by delayed and missed contracts are a necessary nuisance in this endeavor. Through the analysis we performed, we conclude that contract offloading can improve application performance but only when used judiciously. It only makes sense to offload complex contracts (with at least part of the contract requiring linear time computation) due to a significant overhead of data copying.

The future work of this project is to explore better ways to estimate the computational complexity of contracts. We also plan on testing our system on a production-level application, in order to determine it's overall efficacy. Each of these advances in our system will be important in achieving broad applicability and acceptance in the CCA and applications communities.

6. REFERENCES

[1] Allan, B. A., Armstrong, R., Bernholdt, D. E., Bertrand, F., Chiu, K., Dahlgren, T. L., Damevski, K., Elwasif, W. R., Epperly, T. G. W., Govindaraju, M., Katz, D. S., Kohl, J. A., Krishnan, M., Kumfert, G., Larson, J. W., Lefantzi, S., Lewis, M. J., Malony, A. D., McInnes, L. C., Nieplocha, J., Norris, B., Parker, S. G., Ray, J., Shende, S., Windus, T. L., and Zhou, S. A component architecture for high-performance scientific computing. *Intl. J. High-Perf. Computing Appl. 20*, 2 (May 2006), 163–202.

[2] Dahlgren, T. L. Performance-driven interface contract enforcement for scientific components. In *Proceedings of the 10th International Symposium on Component-Based Software Engineering (CBSE-07)* (2007), vol. 4608 of *Lecture Notes in Computer Science*, Springer.

[3] Dahlgren, T. L., and Devanbu, P. T. Improving scientific software component quality through assertions. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications (SE-HPCS '05)* (New York, NY, USA, 2005), ACM.

[4] Kohn, S., Kumfert, G., Painter, J., and Ribbens, C. Divorcing language dependencies from a scientific software library. In *Proceedings of the 10th SIAM Conference on Parallel Processing* (Portsmouth, VA, 2001).

[5] Liblit, B., Aiken, A., Zheng, A. X., and Jordan, M. I. Bug isolation via remote program sampling. In *In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)* (2003), ACM Press, pp. 141–154.

[6] McInnes, L. C., Allan, B. A., Armstrong, R., Benson, S. J., Bernholdt, D. E., Dahlgren, T. L., Diachin, L. F., Krishnan, M., Kohl, J. A., Larson, J. W., Lefantzi, S., Nieplocha, J., Norris, B., Parker, S. G., Ray, J., and Zhou, S. Parallel PDE-based simulations using the Common Component Architecture. In *Numerical Solution of PDEs on Parallel Computers*, A. M. Bruaset and A. Tveito, Eds., vol. 51 of *Lecture Notes in Computational Science and Engineering (LNCSE)*. Springer-Verlag, 2006.

[7] Sokolsky, O., Sammapun, U., Regehr, J., and Lee, I. Runtime verification for wireless sensor network applications. In *Runtime Verification* (2008), B. Finkbeiner, K. Havelund, G. Rosu, and O. Sokolsky, Eds., no. 07011 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.